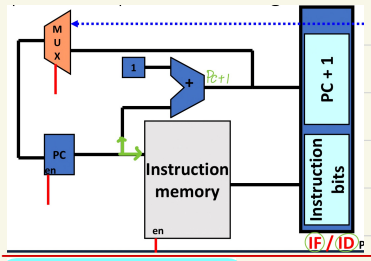
Pipelining

Stage 1: Fetch

任务：① use PC number to read instruction from memory.

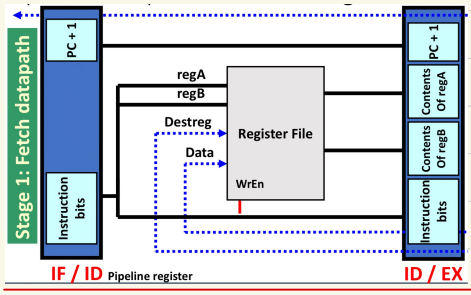
② PC+1. (not consider branches now)



Stage 2: Decode

任务：① Get opcode, regA value, regB value

② keep PC+1, instruction bit



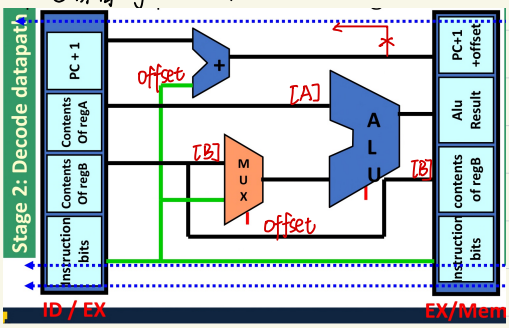
Stage 3: Execute

任务：① ADD/Nor/Beq 来说 input 是 regA value & regB value

② LW/SW 来说 input 是 regA value & offset

③ 计算 PC+1 + offset (现在不知道)

④ 保留 regB value 和 instruction bits

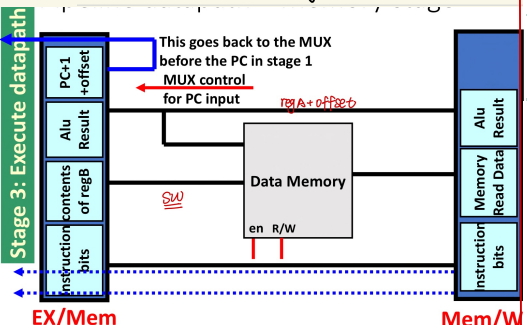


Stage 4: Memory Operation

任务：① 对于 SW 来说将 regB value 存入指定 memory address

② 对于 lw 来说将 memory 特定 address 的值读出.

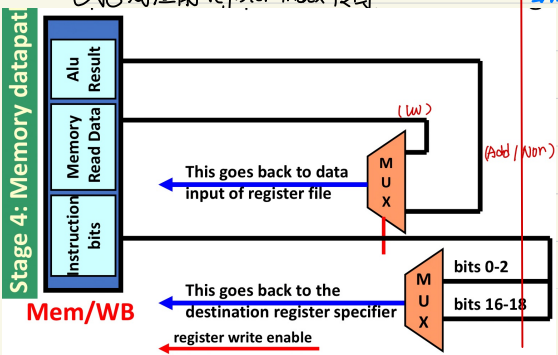
③ 将 PC+1 + offset 传回 IF stage.

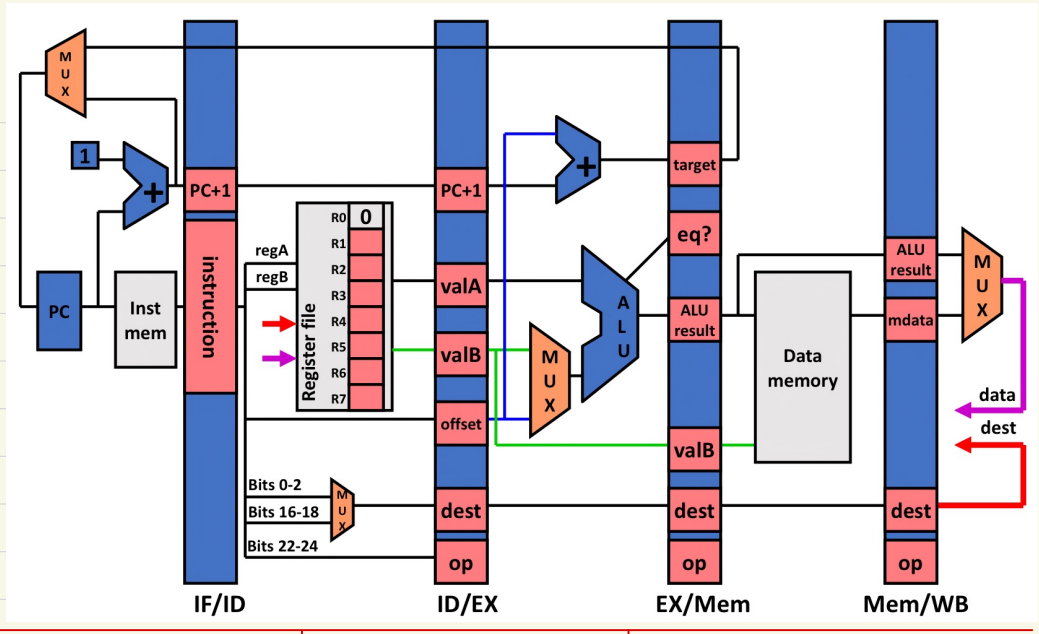


Stage 5: Write back

任务：① 把纵要update的 register value 传回.

② 把对应的 register index 传回





Handling data hazards

I: Avoid all hazards

```
add   1  2  3
noop
noop
nor   3  4  5
```

缺点：

① old programs may not run

② Programs get larger adding "noop"

③ Program execution is slower

Control Hazards

we don't know what to fetch until the memory stage.

1. Avoid
   - Make sure there are no hazards in code
2. Detect and stall
   - Delay fetch until branch resolved

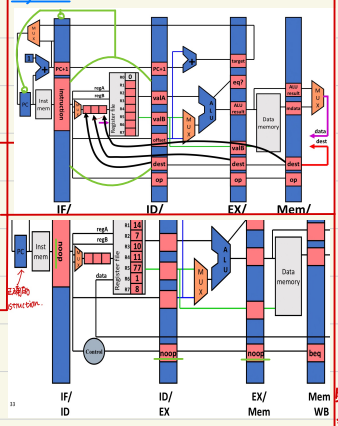
Beq 后面要接3个 noop.

II: Detect and stall until

做法：

- Detect:
  - Compare regA with previous DestRegs
    - 3 bit operand fields
  - Compare regB with previous DestRegs
    - 3 bit operand fields
- Stall:
  - Keep current instructions in fetch and decode
  - Pass a noop to execute

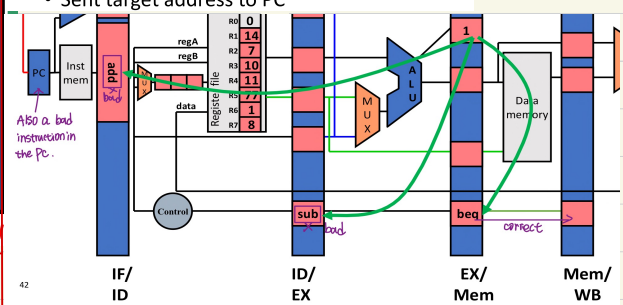
修改：



缺点：① CPI会增加，② beq 的 branch 不是每次都会执行.

3. Speculate and squash-if-wrong

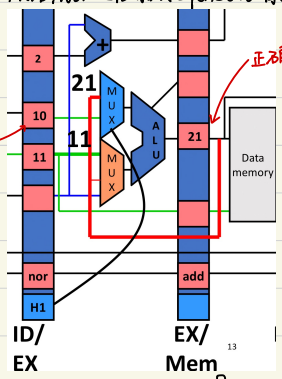
- Speculate: assume not equal
  - Keep fetching from PC+1 until we know that the branch is really taken
- Squash: stop bad instructions if taken
  - Send a noop to Decode, Execute, and Memory
  - Sent target address to PC



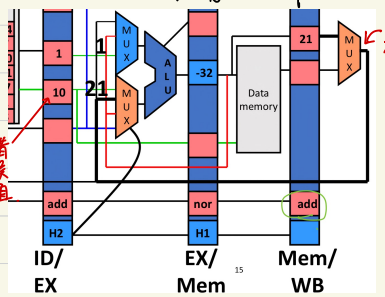
III: Detect and forward

特点：① 不同情况要分开考虑

case 1：ADD/Nor 之后紧跟 dependent 的 instruction

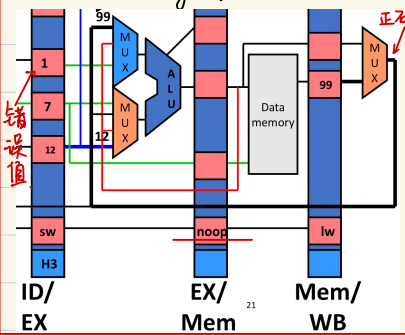


case 2：ADD/Nor 之后隔一个是 dependent 的

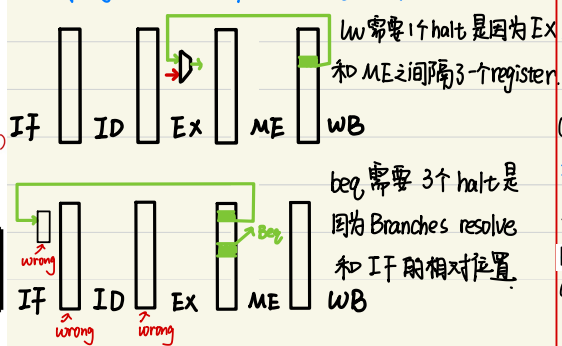


case ③ 紧跟 lw.

因为 lw 必须从 memory 中算值，所以要 halt 一个 "noop"



当改变原有 Stage 数量时计算 lw 和 beq 所需要的新 halt.



lw 需要1个 halt 是因为 Ex 和 ME 之间隔3个 register

beq 需要 3个 halt 是因为 Branches resolve 和 IF 的相对位置

Improve Branch Performance

提到 Beq 需要做的 Prediction：① Instruction 是否是 Beq ② Branch Direction ③ Branch target address

当 Assume taken 时：

使用 Branch Target Buffer (BTB) 来存 Beq 对应的 target

Branch Direction Prediction

① Static
- Always not-taken
- Always taken
- Backward taken, forward not taken (BTFN)

Memory存在的两个问题

①多程序同时 run

②单个程序可能访问的地值过大.

→ Virtual memory

如何 Translate Addresses?

operating system会在 main memory 中 maintain the page table (是一个 data structure)

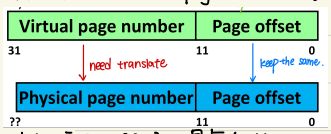
page 的相关特性

(4kB for x86)

- Size of physical page = size of virtual page
- A virtual address consists of
  - A virtual page number
  - A page offset field

改变 page number

- virtual address ⟶ physical address 只需要

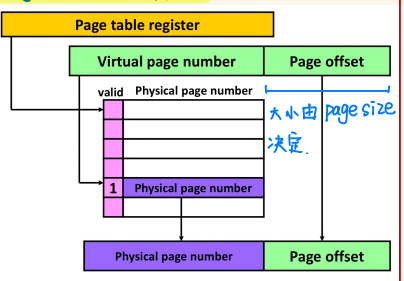


- 为什么要用 page 而不是每个VA都对应一个 physical address.
  ↳ mapping data 占据太多空间
  ⇒ 破坏了 spacial locality

page table 相关特性

- Translate page numbers using **page tables**
- Each process has its own page table
- Page table 本身存在 main memory 且 OS 知道其 physical address. 所以 OS 访问 page table 无需 address translation

page table 的工作原理



大小由 page size 决定.

Virtual Memory Goals

1. **Transparency**
   - Don't need to know how other programs are using memory
2. **Protection**
   - No program can access the data of any other program
3. **Programs not limited by DRAM capacity**
   - Each program can have more data than DRAM size

如何实现 Transparency 和 Protection

Ensure that page table entries across different programs don't contain the same physical page numbers

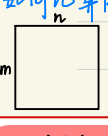
如何实现③

- Use disk as temporary space in case memory capacity is exhausted
  - This temporary space in disk is called swap partition in Linux-based systems
- We can mark a page table entry as "Invalid", indicating that the data for that page doesn't exist in main memory, but instead is located on the disk
- Looking up a page table entry that corresponds to disk is called a **page fault**

出现 page fault 的解决方法.

1. Stop this process
2. Pick page to replace
3. Write back data
4. Get referenced page
5. Update page table
6. Reschedule process

如何计算 page table 大小



n: physical page number 占多少 bytes.

m: virtual page number 有多少.

对于 one-level page table 而言. page table 大小固定

Multi-level page-tables

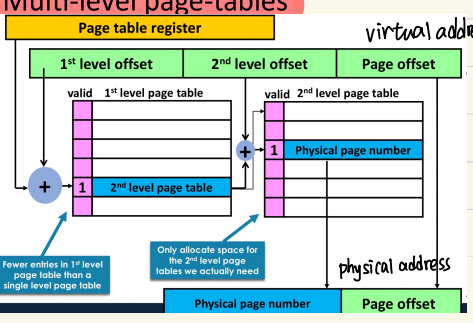
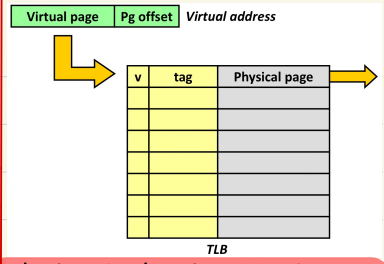


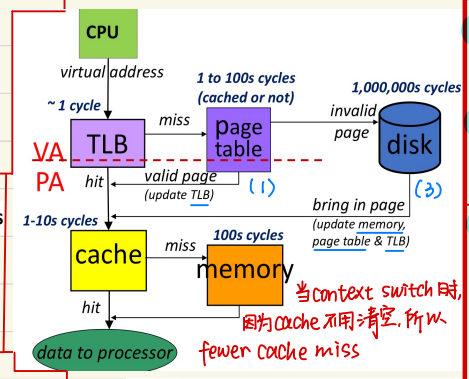
Table Look-aside Buffers (TLB)



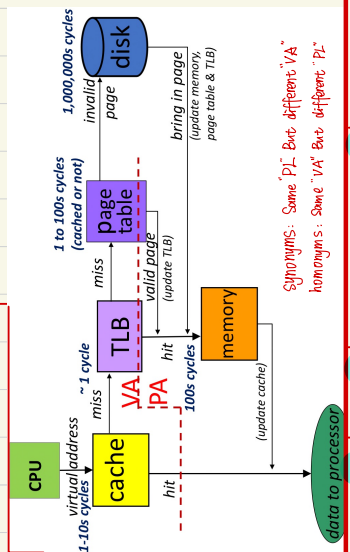
Placing Caches in a VM System

访问 Cache 所使用的 address type

- Physical address (after VM translations).
  - We have to wait for the translation; slower.
- Virtual address (before VM translation).
  - Faster access.
  - More complex.



当 context switch 时, 因为 cache 不用清空. 所以 fewer cache miss



为保证 OS 可以 modify page table register 和 update page table data

但 user program 不行:

Executive / Supervisor / kernel level

user level

Cache Performance

① Cache Size

- bigger can exploit temporal locality better
- Cache size 大到一定程度对 hit rate 的提高就不在显著.
- Cache size 越大 access time 越慢.

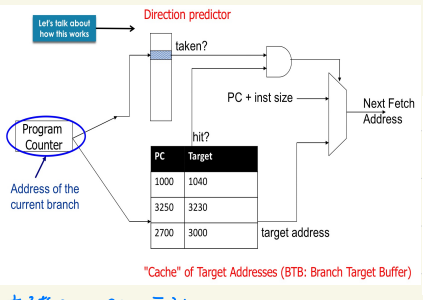
② Block size

- Too small blocks
  - don't exploit spatial locality well
  - have larger tag overhead
- Too large blocks
  - too few total # of blocks

③ Associativity

- Larger associativity
  - lower miss rate, less variation among programs
  - diminishing returns
- Smaller associativity
  - lower cost
  - faster hit time

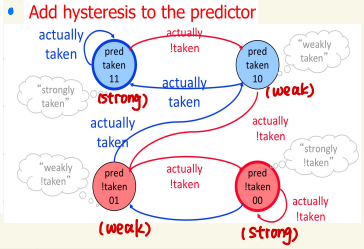
hit r:



② Dynamic

- Last time predictor

- Add hysteresis to the predictor



存储 Data 的硬件:

1: SRAM	2: DRAM	3: Disks
• need constant power	• need constant power	• Don't need power
• Fast: ~1 ns read time	• Slower: ~50 ns access time	• slow
• small & Expensive	• Bigger & Cheaper	• Cheapest

- **Average Latency:** $cache\ latency + (memory\ latency \cdot miss\ rate)$

reduce overhead

1	3	160	170
V	tag	data (block)	

Overhead = (3+1) / 16 = 25%

Overhead = (非 data 部分 bit 数) / (data 部分 bit 数)

tag 的 bit 数量取决于 memory 能被分为多少个 block

Figuring out the tag

$tag = \left\lfloor \frac{addr}{block\ size} \right\rfloor$ 剩下的是 block offset

about stores

① 若该 memory location 已在 Cache 中.

- 只在 Cache 中改. (write back)
- 在 Cache 中改, 也在 memory 中改. (write through policy)

② 若该 memory location 不在 Cache 中

- 直接改 memory.
- 先将该 memory location 读入 Cache, 再改 Cache 中值. 最终改 memory 中值. (allocate-on-write policy)

memory references (write-through)

① Total Read: #miss 的 read × block 大小

② Total write: # Load × 1 byte

若是 (write back)

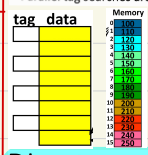
Total write: # dirty bit = 1 × block 大小

write-back

V	d	tag	data
0			

Fully-associative caches

- Any memory location can be copied to any cache line.
- We check every cache tag to determine whether the data is in the cache.
- Parallel tag searches are expensive and can be slow



Address:

tag	block offset
3 bits	1 bit

A block can go to any location

Direct mapped caches

- Direct mapped caches partition memory into as many regions as there are cache lines
- Each memory region maps to a **single cache line** in which data can be placed
- Now only **one tag** needs to be checked – the one associated with the region the reference is in

A block can go to only one location

Address:

tag	line index	block offset
1 bit	2 bits	1 bit

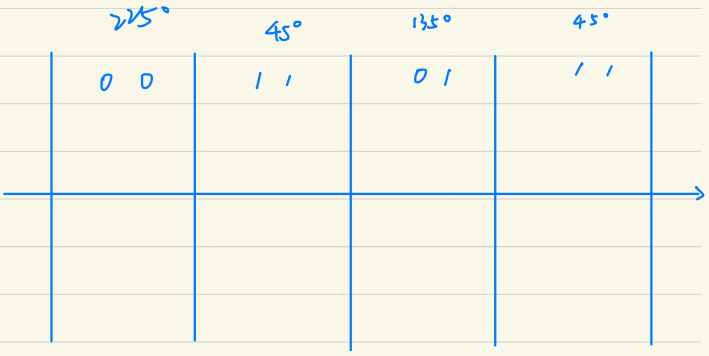
Set-associative cache

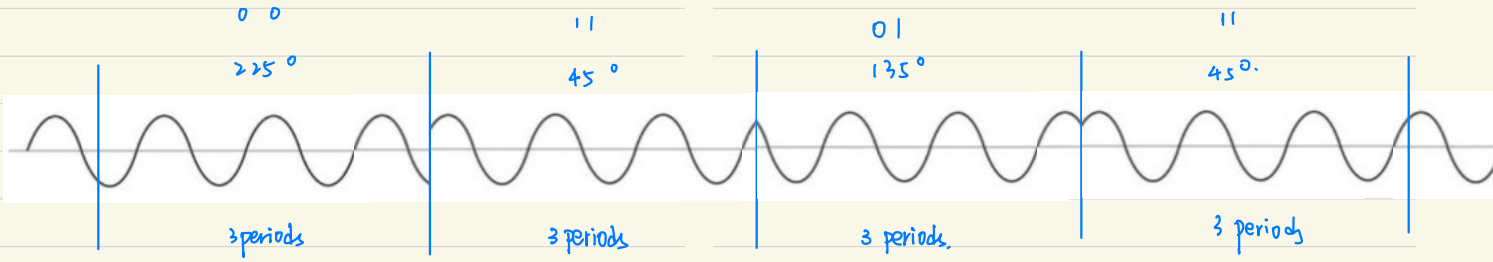
- # block offset bits = $\log_2$(block size)
- # set index bits = $\log_2$(# of sets)
- # tag bits = rest of address bits

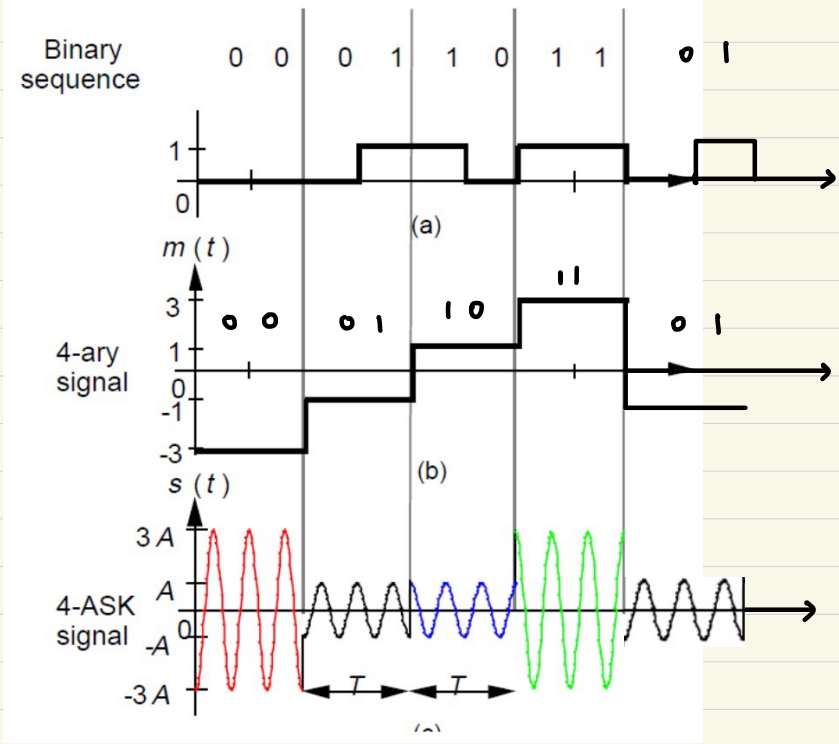
tag	set index	block offse
2 bit	1 bits	1 bit

Classifying Cache Misses

- **Compulsory miss**
  - We've never accessed this data before
- **Capacity miss**
  - Cache is not large enough to hold all the data
  - May have been avoided if we used a bigger cache
- **Conflict miss**
  - Cache is large enough to hold data, but was replaced due to overly restrictive associativity
  - May have been avoided if we used a higher-associative cache


225°
45°
135°
45°
0 0
1 1
0 1
1 1


0 0
225°
3 periods
1 1
45°
3 periods
0 1
135°
3 periods
1 1
45°
3 periods


Binary sequence
0 0 0 1 1 0 1 1 0 1
1
0
(a)
m (t)
3
1
0
-1
-3
4-ary signal
0 0
0 1
1 0
1 1
0 1
(b)
s (t)
3 A
A
0
-A
-3 A
4-ASK signal
T
T